

February 5, 2021

# 1 Alessandro Pomponio

## 1.1 Matricola 0000920265

1. load the data and separate in X all the columns but the last one, in y the last column, then produce a `pairplot` of X and decide which pair of columns is most interesting for a 2d scatterplot, ad produce the scatterplot (5pt)
2. find the best clustering scheme for X with a method of your choice, plot `ssd` and global `silhouette` index for an appropriate range of parameters and show the chosen hyperparameter(s) (5pt)
3. consider carefully the number of clusters, simple optimisation of the silhouette will not be enough, consider also the elbow plot and decide visually the best number of clusters
4. fit the clustering scheme to `y_km`, then produce the confusion matrix comparing y and `y_km` with `sklearn.metrics.confusion_matrix`, the resulting confusion matrix must be “sorted” using the function `max_diag` provided below, producing the final confusion matrix `cm_km` (5pt)
5. in a comment explain why function `max_diag` is useful (2pt)
6. compute the accuracy `a_km` of `y_km` versus y as the ratio the sum of the main diagonal of `cm_km` and the number of samples in X (2pt)
7. rescale X using `sklearn.preprocessing.MinMaxScaler`, producing the scaled dataset `X_mms` (3pt)
8. repeat point 3 and 5 above, fitting `X_mms` to `y_km_mms` and producing the confusion matrix `cm_km_mms` reordered with `max_diag` and the accuracy `a_km_mms` as above (3pt)

- 1.1.1 1. load the data and separate in X all the columns but the last one, in y the last column, then produce a `pairplot` of X and decide which pair of columns is most interesting for a 2d scatterplot, ad produce the scatterplot (5pt)

```
[75]: # Imports
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score
from sklearn.metrics import confusion_matrix
from sklearn.preprocessing import MinMaxScaler
from plot_clusters import plot_clusters
from max_diag import max_diag
```

```

# Variables
file_name = 'seeds_dataset.txt'
separator = '\t'
random_state = 42
target = 7

# Directives
%matplotlib inline
np.random.seed(random_state)

```

We use numpy to load the dataset, then convert it to a Pandas DataFrame to operate on it. We'll also print the head of the dataframe to have a look at it.

```

[46]: text = np.loadtxt(file_name, delimiter = separator)
      df = pd.DataFrame(text)
      df.head()

```

```

[46]:
      0      1      2      3      4      5      6      7
0  15.26  14.84  0.8710  5.763  3.312  2.221  5.220  0.0
1  14.88  14.57  0.8811  5.554  3.333  1.018  4.956  0.0
2  14.29  14.09  0.9050  5.291  3.337  2.699  4.825  0.0
3  13.84  13.94  0.8955  5.324  3.379  2.259  4.805  0.0
4  16.14  14.99  0.9034  5.658  3.562  1.355  5.175  0.0

```

We now separate the feature matrix from the target column

```

[47]: X = df.drop(target, axis = 1)
      y = df[target]

      print(f"The feature matrix has {X.shape[0]} rows and {X.shape[1]} columns")

```

The feature matrix has 210 rows and 7 columns

```

[69]: print(f"There are {len(np.unique(y))} unique classes")

```

There are 3 unique classes

We will now produce a pairplot to look for possible interesting patterns.

```

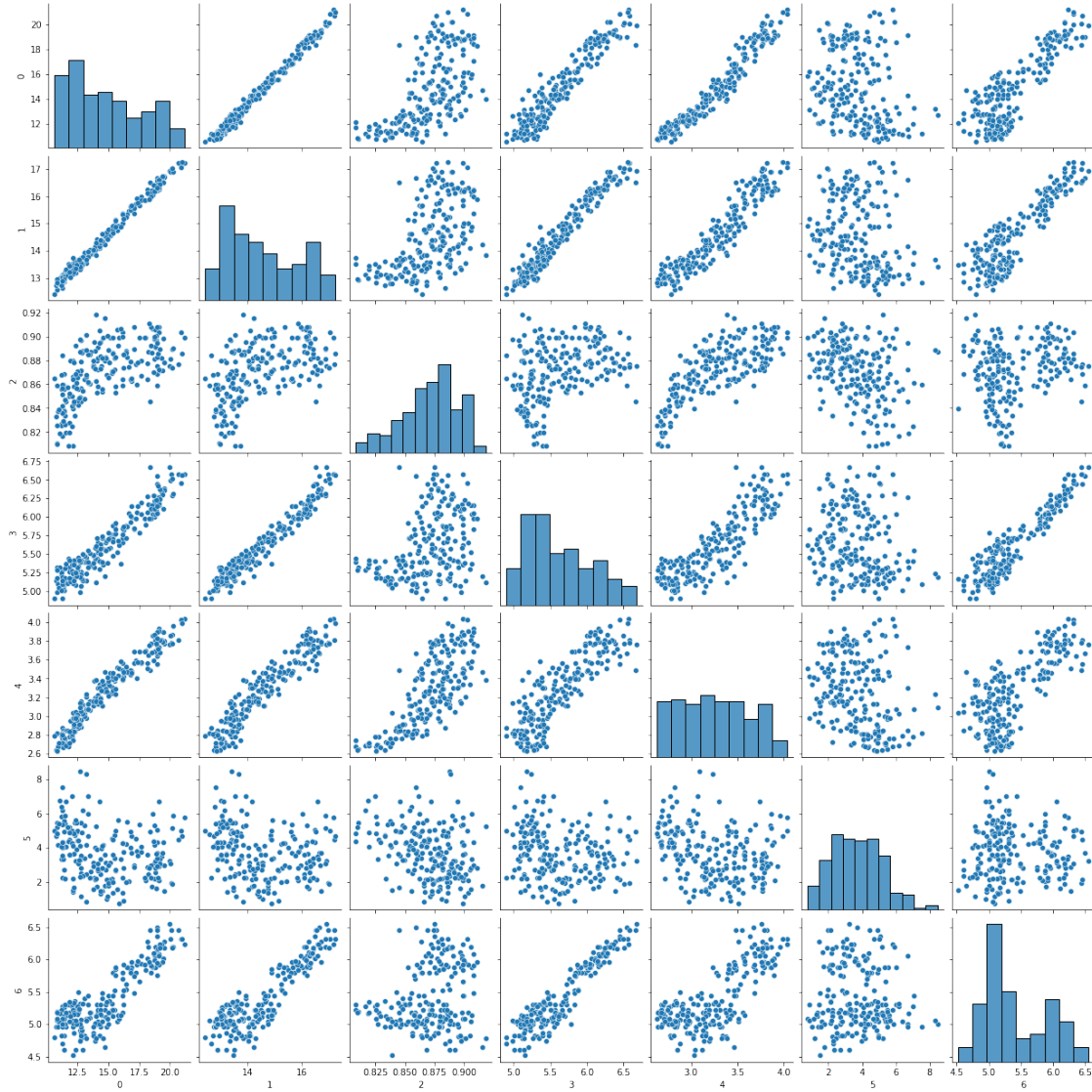
[48]: sns.pairplot(X)

```

```

[48]: <seaborn.axisgrid.PairGrid at 0x22e00d38e50>

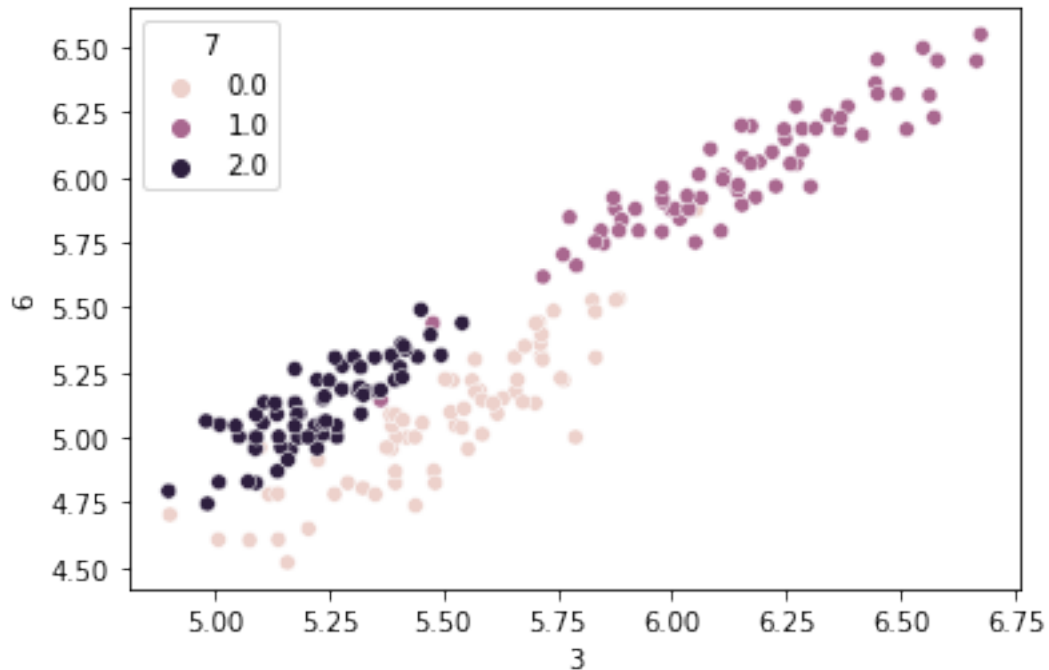
```



Judging by the pairplots, we choose the columns with indexes 3,6, as they seem to create three distinct clusters, one for each of the classes that we have. We will plot a scatterplot using those two columns.

```
[68]: focus = [3,6]
      sns.scatterplot(x = focus[0], y = focus[1], data = df, hue = target)
```

```
[68]: <AxesSubplot:xlabel='3', ylabel='6'>
```



1.1.2 2. find the best clustering scheme for X with a method of your choice, plot ssd and global silhouette index for an appropriate range of parameters and show the chosen hyperparameter(s) (5pt)

In order to find a clustering scheme, we will use K-means with the elbow method, ranging from 2 to 10 clusters

```
[50]: # Range of possible clusters
k_range = range(2,11)

# Distortion and Silhouette Score as measures
distortions = []
silhouette_scores = []

for i in k_range:

    km = KMeans(n_clusters = i,
                init = 'k-means++',
                n_init = 10,
                max_iter = 300,
                random_state = random_state)

    y_km = km.fit_predict(X)
    distortions.append(km.inertia_)
    silhouette_scores.append(silhouette_score(X,y_km))
```

Plot distortion and silhouette indexes

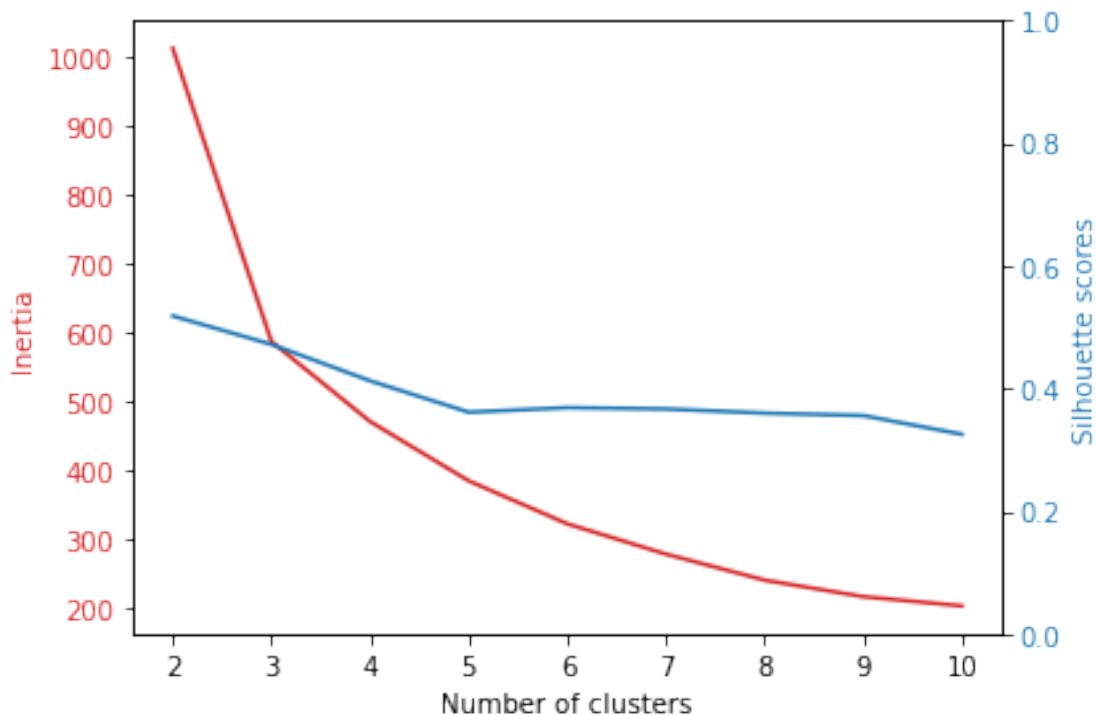
```
[51]: fig, ax1 = plt.subplots()

color = 'tab:red'
ax1.set_xlabel('Number of clusters')
ax1.set_ylabel('Inertia', color=color)
ax1.plot(k_range, distortions, color=color)
ax1.tick_params(axis='y', labelcolor=color)

ax2 = ax1.twinx() # instantiate a second axes that shares the same x-axis

color = 'tab:blue'
ax2.set_ylabel('Silhouette scores', color=color) # we already handled the x-label with ax1
ax2.plot(k_range, silhouette_scores, color=color)
ax2.tick_params(axis='y', labelcolor=color)
ax2.set_ylim(0,1) # the axis for silhouette is [0,1]

fig.tight_layout() # otherwise the right y-label is slightly clipped
plt.show()
```



The silhouette scores plot tells us that we obtain the best result with 2 clusters

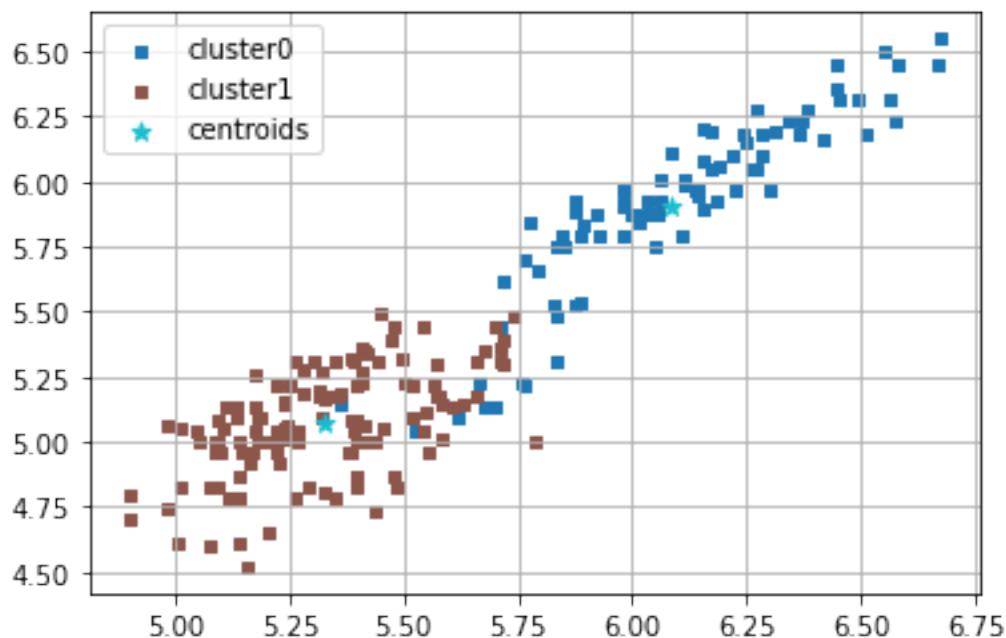
```
[52]: silhouette_best_k = 2
```

### 1.1.3 3. consider carefully the number of clusters, simple optimisation of the silhouette will not be enough, consider also the elbow plot and decide visually the best number of clusters

The elbow plot seems to suggest that there should be 3 clusters instead of 2. We will plot both clustering schemes and choose visually

```
[53]: # First attempt with 2 clusters
km = KMeans(n_clusters = silhouette_best_k,
            init = 'k-means++',
            n_init = 10,
            max_iter = 300,
            tol = 1e-04,
            random_state = random_state)

y_km = km.fit_predict(X)
plot_clusters(X.to_numpy(), y_km, dim=(focus[0],focus[1]), points = km.
             →cluster_centers_)
```



```
[54]: # Second attempt with 3 clusters
elbow_best_k = 3

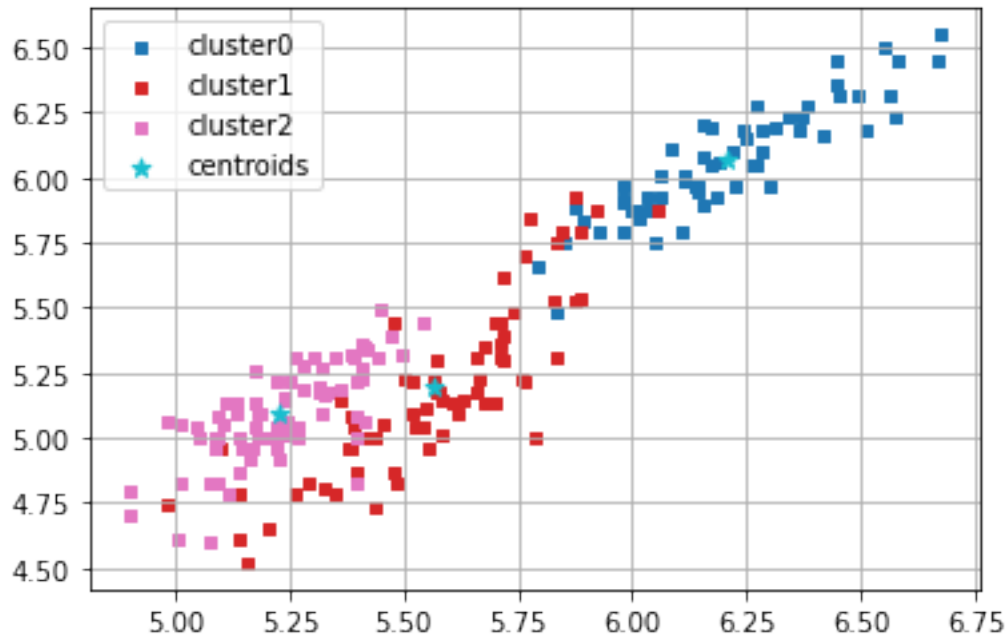
km = KMeans(n_clusters = elbow_best_k,
            init = 'k-means++',
```

```

n_init = 10,
max_iter = 300,
tol = 1e-04,
random_state = random_state)

y_km = km.fit_predict(X)
plot_clusters(X.to_numpy(), y_km, dim=(focus[0],focus[1]), points = km.
    →cluster_centers_)

```



The clustering scheme with three clusters seems to indeed be the better one of the two

```
[64]: best_k = 3
```

**1.1.4 4. fit the clustering scheme to `y_km`, then produce the confusion matrix comparing `y` and `y_km` with `sklearn.metrics.confusion_matrix`, the resulting confusion matrix must be “sorted” using the function `max_diag` provided below, producing the final confusion matrix `cm_km` (5pt)**

Let's start by fitting the clustering scheme to `y_km` and producing the confusion matrix

```
[65]: km = KMeans(n_clusters = best_k,
    init = 'k-means++',
    n_init = 10,
    max_iter = 300,
    tol = 1e-04,
    random_state = random_state)

```

```
y_km = km.fit_predict(X)
km.fit(X, y = y_km)
```

```
[65]: KMeans(n_clusters=3, random_state=42)
```

```
[72]: cm = confusion_matrix(y, y_km)
      print(cm)
```

```
[[ 1 60  9]
 [60 10  0]
 [ 0  2 68]]
```

We now have a look at the function `max_diag`

```
[58]: help(max_diag)
```

Help on function `max_diag` in module `max_diag`:

`max_diag(sq_arr)`

Given a square matrix produces another squared matrix with the same contents,

but the columns are re-ordered in order to have the highest values in the main diagonal

Parameter: `sq_arr` - a squared matrix

Example:

In [1]: `import numpy as np`  
`max_diag(np.array([[1,10],[20,2]]))`

Out[1]: `array([[10., 1.],`  
 `[ 2., 20.]])`

This function is useful to reorder a confusion matrix when the two label vectors

have different codings

We can now print the sorted confusion matrix

```
[71]: cm_km = max_diag(cm)
      print(cm_km)
```

```
[[60.  1.  9.]
 [10. 60.  0.]
 [ 2.  0. 68.]]
```

### 1.1.5 5. in a comment explain why function `max_diag` is useful (2pt)

As the help function says, **the function is useful to reorder a confusion matrix when the two label vectors have different codings**



**1.1.6 6. compute the accuracy `a_km` of `y_km` versus `y` as the ratio the sum of the main diagonal of `cm_km` and the number of samples in `X` (2pt)**

```
[74]: a_km = np.diagonal(cm_km).sum() / X.shape[0]
      print(f"The accuracy was {a_km * 100:.2f}%")
```

The accuracy was 89.52%

**1.1.7 7. rescale `X` using `sklearn.preprocessing.MinMaxScaler`, producing the scaled dataset `X_mms` (3pt)**

```
[77]: scaler = MinMaxScaler()
      X_mms = scaler.fit_transform(X)
      print(X_mms)
```

```
[[0.44098206 0.50206612 0.5707804 ... 0.48610121 0.18930164 0.34515017]
 [0.40509915 0.44628099 0.66243194 ... 0.50106914 0.03288302 0.21516494]
 [0.34938621 0.34710744 0.87931034 ... 0.50392017 0.25145302 0.1506647 ]
 ...
 [0.24645892 0.25826446 0.7277677 ... 0.42908054 0.98166664 0.26440177]
 [0.11803588 0.16528926 0.39927405 ... 0.14682823 0.36834441 0.25849335]
 [0.16147309 0.19214876 0.54718693 ... 0.24518888 0.63346292 0.26784835]]
```

**1.1.8 8. repeat point 3 and 5 above, fitting `X_mms` to `y_km_mms` and producing the confusion matrix `cm_km_mms` reordered with `max_diag` and the accuracy `a_km_mms` as above (3pt)**

```
[78]: y_km_mms = km.fit_predict(X_mms)
      km.fit(X_mms, y = y_km_mms)

      cm_km_mms = max_diag(confusion_matrix(y, y_km_mms))
      a_km_mms = np.diagonal(cm_km_mms).sum() / X.shape[0]
      print(f"The accuracy was {a_km_mms * 100:.2f}%")
```

The accuracy was 89.52%